Qiao Liu 146800980
Duc Nguyen 160246450
Rob Fedorowicz 170783690
Stevie Gallow 170405240
Simon Arcila 130804570

CP468 Assignment 2: Sudoku As A CSP

Introduction

A sudoku puzzle is a logic-based, combinatorial number-placement puzzle. The objective is to fill a 9×9 grid with digits so that each column, each row, and each of the nine 3×3 subgrids that compose the grid (also called "boxes", "blocks", or "regions") contain all of the digits from 1 to 9. It can be considered as a CSP because there are 81 variables, one variable for each node, among which those empty nodes have the domain {1,2,3,4,5,6,7,8,9}, while those prefilled nodes contain only one value in the domain. Moreover, the sudoku puzzle is a 9-consistency CSP in that there are 27 constraints which are 9 rows, 9 columns and 9 boxes, and each constraint involves 9 nodes. In this assignment, we will solve the sudoku puzzle by using AC-3, forward checking and backtracking search algorithms.

Constraints Handling

AC-3 is used to make the entire CSP arc-consistent on the condition that the CSP only contains binary constraints or unary constraints. To solve this puzzle in AC3, we need to transform all the higher-order constraints to binary constraints. The constraint tells us that each of 9 variables can choose one integer from 1 to 9. While each variable can pick one integer from 1 to 9, no two variables can have the same integer.

$$
\begin{gathered}
\text{Alldiff}(A, B, C, D, E, F, G, H, I) = \\
\text{Diff}(A, B) + \text{Diff}(A, C) + \text{Diff}(A, D) + \text{Diff}(A, E) + \text{Diff}(A, F) + \text{Diff}(A, G) + \text{Diff}(A, H) + \text{Diff}(A, I) \\
+\text{Diff}(B, C) + \text{Diff}(B, D) + \text{Diff}(B, E) + \text{Diff}(B, F) + \text{Diff}(B, G) + \text{Diff}(B, H) + \text{Diff}(B, I) \\
+\text{Diff}(C, D) + \text{Diff}(C, E) + \text{Diff}(C, F) + \text{Diff}(C, G) + \text{Diff}(C, H) + \text{Diff}(C, I) \\
+\text{Diff}(D, E) + \text{Diff}(D, F) + \text{Diff}(D, G) + \text{Diff}(D, H) + \text{Diff}(D, I) \\
+\text{Diff}(E, F) + \text{Diff}(E, G) + \text{Diff}(E, H) + \text{Diff}(E, I) \\
+\text{Diff}(F, G) + \text{Diff}(F, H) + \text{Diff}(F, I) \\
+\text{Diff}(G, H) + \text{Diff}(G, I) \\
+\text{Diff}(H, I)
\end{gathered}
$$

The formula above shows that one constraint can be transformed to 36 Diff(A,B) constraints. Therefore, we will add 1944 (27 x 36 x 2) arcs into ArrayList<Arc> arcs before AC-3 starts. Notice that Arch(node1,node2) is different with Arch(node2,node1).

AC-3

AC-3 is a way of reducing the domains of variables by making them arc-consistent before searching. In the previous section, we have insert all arcs into a set of arcs. In AC-3, we remove one arc from the set of arcs each time then check node1's arc-consistency with respect to node2. Values will be removed from node1’s domain if they are also in node2’s domain. If node1’s domain contains no value after this process, we can assume that the puzzle has no solution. Otherwise we add all Arch(node3,node1) into set arcs where node3 has constraint with node1 and node3 is not node2 (resulted from addNeighborArcsExceptB). We continue the whole step until the set is empty or one node's domain is empty. In some situations, we can find a solution when AC-3 is finished. Here is one example: after 320 steps of shrinking domains we find a solution.

```
// output when running
// new AC3(Node.getNodes("data0"));
// Test.VERBOSE = true;
loading puzzle from data0...
[0][0][0][0][0][0][6][0][0]
[8][0][4][0][0][0][0][0][7]
[0][0][5][0][0][3][0][9][0]
[0][0][1][0][9][8][0][5][2]
[6][9][2][0][0][7][3][0][0]
[4][5][0][1][0][0][0][0][6]
[5][0][0][3][6][0][2][4][9]
[2][4][0][8][7][0][1][0][3]
[3][0][6][2][4][9][7][8][5]
AC3...
step 1 set.size():1933 AC:(0,0)(0,6) (0,0).remove(6) domain:[12345789]
step 2 set.size():1938 AC:(0,1)(0,6) (0,1).remove(6) domain:[12345789]
...
step 319 set.size():770 AC:(0,1)(1,1) (0,1).remove(2) domain:[7]
step 320 set.size():392 AC:(0,7)(0,2) (0,7).remove(3) domain:[2]
[9][7][3][5][8][4][6][2][1]
[8][2][4][9][1][6][5][3][7]
[1][6][5][7][2][3][8][9][4]
[7][3][1][6][9][8][4][5][2]
[6][9][2][4][5][7][3][1][8]
[4][5][8][1][3][2][9][7][6]
[5][8][7][3][6][1][2][4][9]
[2][4][9][8][7][5][1][6][3]
[3][1][6][2][4][9][7][8][5]
total running time:21ms
```

Forward Checking

Like AC-3, forward checking is also a way of inferencing reductions in variables' domains before searching. Whenever a variable X is assigned, the forward checking process establishes arc-consistency for it. For each unassigned variable Y that is connected to X by a constraint, we will delete any value from Y’s domain that is inconsistent with the value chosen for X. If Y’s domain is empty, one inference can be made that there is no solution. The forward checking method that we use is slightly different. Instead of checking the Ys' arc-consistencies with the newly assigned X, we replace X with all variables containing only one value. It may take more steps for executing, but there is no need to check whether a variable with one value is given by the puzzle or assigned by our assumption. When forward checking is applied after one puzzle is given, the result is the same with that of AC-3 because they both make all variables arc-consistent. However, since the former one has recursion and pruning, we choose to use it during backtracking search step.

```
// output when running
// new AC3(Node.getNodes("data6"));
// new Backtracking(Node.getNodes("data6"));
// Test.VERBOSE = false;
loading puzzle from data6...
[0][7][0][8][0][0][0][0][0]
[2][0][0][3][0][6][0][7][0]
[0][9][8][0][2][0][1][3][0]
[0][5][0][0][0][0][9][8][7]
[8][0][0][0][7][0][0][4][1]
[9][0][7][0][0][0][0][5][2]
[0][0][9][0][6][0][5][0][3]
[7][0][0][1][0][4][0][6][0]
[0][0][0][0][0][3][7][0][4]
AC3...
[13456][7][13456][8][1459][159][246][29][569]
[2][14][145][3][1459][6][48][7][589]
[456][9][8][457][2][57][1][3][56]
[1346][5][12346][246][134][12][9][8][7]
[8][236][236][2569][7][259][36][4][1]
[9][1346][7][46][1348][18][36][5][2]
[14][1248][9][27][6][278][5][12][3]
[7][238][235][1][589][4][28][6][89]
[156][1268][1256][259][589][3][7][129][4]
first time of forward checking...
[13456][7][13456][8][1459][159][246][29][569]
[2][14][145][3][1459][6][48][7][589]
[456][9][8][457][2][57][1][3][56]
```

```
[1346][5][12346][246][134][12][9][8][7]
[8][236][236][2569][7][259][36][4][1]
[9][1346][7][46][1348][18][36][5][2]
[14][1248][9][27][6][278][5][12][3]
[7][238][235][1][589][4][28][6][89]
[156][1268][1256][259][589][3][7][129][4]
```

Backtracking

Backtracking search uses a depth-first search to choose values for one variable at a time and backtracks when a variable has no legal values left to assign. Here we can apply heuristics when selecting variables and values. In the source code, the getHeuristicNode() function uses the minimum-remaining-values (MRV) heuristic, getting one unassigned node (node's domain > 1) with the minimum domain size. There is no need to apply degree heuristic for choosing the first node to expand because in the sudoku puzzle, degree heuristic and MRV heuristic are the same node with the most constraints with other unassigned nodes are always the one having the minimum remaining values. Moreover, every time when we remove one value from one variable's domain, we put this variable and the removed value as a record into a stack. Then we use the size of the stack to represent current state. Next time when we want to backtrack to former state, we use the backtrackState(state) function to remove those records exceeding the recorded size of the stack and then put values back into their corresponding variables' domains. Backtracking search starts from one variable, it assigns one value to the variable then does forward checking each time. When forward checking finds no error, it will use recursion to search on another node. Otherwise it will remove this value from current variable's domain and do forward check on another value.

Input & Output

We have collected 10 sudoku puzzles and they shall be stored in text files from data0 to data9. The empty variables are assigned to 0. In Test.java we will use the static method Node.getNodes(filename) to read the puzzle and transfer it into a Node[][]. During this step, we will initialize each node with its row index, column index, box index and domain. We will then call new AC3(Node[][]) and new Backtracking(Node[][]) to solve the puzzle. AC3.java will first use AC-3 to enforce arc consistency on each node then call backtracking search if the puzzle is not solved while Backtracking.java will first use forward checking on all nodes then call backtracking search.

Below is the output of new AC3(Node.getNodes("data9")):

```
loading puzzle from data9...
[0][0][5][3][0][0][0][0][0]
[8][0][0][0][0][0][0][2][0]
[0][7][0][0][1][0][5][0][0]
[4][0][0][0][0][5][3][0][0]
[0][1][0][0][7][0][0][0][6]
[0][0][3][2][0][0][0][8][0]
[0][6][0][5][0][0][0][0][9]
[0][0][4][0][0][0][0][3][0]
[0][0][0][0][0][9][7][0][0]
AC3...
[1269][249][5][3][24689][24678][14689][14679][1478]
[8][349][169][4679][4569][467][1469][2][1347]
[2369][7][269][4689][1][2468][5][469][348]
[4][289][26789][1689][689][5][3][179][127]
[259][1][289][489][7][348][249][459][6]
[5679][59][3][2][469][146][149][8][1457]
[1237][6][1278][5][2348][123478][1248][14][9]
[12579][2589][4][1678][268][12678][1268][3][1258]
[1235][2358][128][1468][23468][9][7][1456][12458]
backtracking...
[1][4][5][3][2][7][6][9][8]
[8][3][9][6][5][4][1][2][7]
[6][7][2][9][1][8][5][4][3]
[4][9][6][1][8][5][3][7][2]
[2][1][8][4][7][3][9][5][6]
[7][5][3][2][9][6][4][8][1]
[3][6][7][5][4][2][8][1][9]
[9][8][4][7][6][1][2][3][5]
[5][2][1][8][3][9][7][6][4]
total running time:70ms
```

Below is the output of new Backtracking(Node.getNodes("data9")) :

```
loading puzzle from data9...
[0][0][5][3][0][0][0][0][0]
[8][0][0][0][0][0][0][2][0]
[0][7][0][0][1][0][5][0][0]
[4][0][0][0][0][5][3][0][0]
[0][1][0][0][7][0][0][0][6]
[0][0][3][2][0][0][0][8][0]
[0][6][0][5][0][0][0][0][9]
[0][0][4][0][0][0][0][3][0]
[0][0][0][0][0][9][7][0][0]
first time of forward checking...
[1269][249][5][3][24689][24678][14689][14679][1478]
[8][349][169][4679][4569][467][1469][2][1347]
[2369][7][269][4689][1][2468][5][469][348]
[4][289][26789][1689][689][5][3][179][127]
[259][1][289][489][7][348][249][459][6]
[5679][59][3][2][469][146][149][8][1457]
[1237][6][1278][5][2348][123478][1248][14][9]
[12579][2589][4][1678][268][12678][1268][3][1258]
[1235][2358][128][1468][23468][9][7][1456][12458]
backtracking...
[1][4][5][3][2][7][6][9][8]
[8][3][9][6][5][4][1][2][7]
[6][7][2][9][1][8][5][4][3]
[4][9][6][1][8][5][3][7][2]
[2][1][8][4][7][3][9][5][6]
[7][5][3][2][9][6][4][8][1]
[3][6][7][5][4][2][8][1][9]
[9][8][4][7][6][1][2][3][5]
[5][2][1][8][3][9][7][6][4]

total running time:58ms
```

Based on the results, we can see that the result after AC-3 and after the first time of forward checking are the same. Two executions are the same during the later backtracking search step while the total running times are different, this is because forward checking uses recursion and pruning thus performs better than AC-3. We can also set the static boolean VERBOSE=true in Test.java to show detailed records of steps in AC-3 and backtracking search, getting output like this:

```
...
AC3...
step 1 set.size():1941 AC:(0,0)(0,2) (0,0).remove(5) domain:[12346789]
step 2 set.size():1958 AC:(0,0)(0,3) (0,0).remove(3) domain:[1246789]
step 3 set.size():1965 AC:(0,1)(0,2) (0,1).remove(5) domain:[12346789]
...
step 302 set.size():5766 AC:(6,7)(8,6) (6,7).remove(7) domain:[14]
step 303 set.size():5778 AC:(7,6)(6,8) (7,6).remove(9) domain:[1268]
step 304 set.size():5769 AC:(8,8)(7,7) (8,8).remove(3) domain:[12458]
step 305 set.size():5787 AC:(7,8)(8,6) (7,8).remove(7) domain:[1258]
backtracking...
step 1 try 5 for (5,1)
step 2 try 2 for (4,0)
step 3 try 8 for (3,1)
...
step 81 try 6 for (0,4)
step 82 try 1 for (0,0)
step 83 try 9 for (0,7)
```

Data Files

data0

```
0 0 0 0 0 0 6 0 0
8 0 4 0 0 0 0 0 7
0 0 5 0 0 3 0 9 0
0 0 1 0 9 8 0 5 2
6 9 2 0 0 7 3 0 0
4 5 0 1 0 0 0 0 6
5 0 0 3 6 0 2 4 9
2 4 0 8 7 0 1 0 3
3 0 6 2 4 9 7 8 5
```

data1

```
0 0 0 6 4 0 0 1 0
5 0 0 0 0 0 0 9 0
3 0 0 0 0 0 0 0 0
7 0 0 0 0 0 0 0 5
0 0 1 4 0 0 0 0 0
0 0 0 0 0 0 8 0 3
8 0 0 0 0 3 0 0 0
0 0 0 2 0 0 0 0 0
0 0 4 0 0 0 0 6 0
```

data2

```
0 0 0 8 9 0 0 0 0
0 0 7 0 0 0 2 6 0
0 0 0 0 0 5 0 0 0
0 0 0 0 5 3 4 0 0
9 0 0 7 0 0 0 0 0
0 0 0 0 0 0 1 0 0
0 0 1 0 0 2 0 0 0
0 0 0 0 0 0 0 0 8
0 0 0 6 0 0 0 9 0
```

data3

```
0 0 0 0 1 0 0 0 0
0 0 8 0 0 6 0 0 4
0 2 0 0 0 0 0 9 0
0 0 0 0 0 0 2 7 0
3 0 0 0 0 0 0 0 0
6 0 4 0 0 5 0 0 0
0 0 0 0 7 0 0 0 0
0 0 5 0 0 0 0 0 3
0 0 0 9 2 0 0 0 0
```

data4

```
0 0 4 0 0 0 0 0 5
0 0 6 8 9 0 0 0 0
0 0 0 0 3 0 0 0 0
0 0 0 0 0 0 4 9 0
5 2 0 0 0 7 0 0 0
0 0 0 0 0 0 0 0 0
3 0 0 2 0 0 0 0 0
0 0 0 0 0 5 0 0 7
0 0 9 0 0 0 0 8 0
```

data5

```
6 0 9 0 0 0 3 0 0
0 0 0 0 0 3 0 0 6
0 3 0 1 0 0 8 5 2
0 0 0 8 0 0 6 0 1
5 6 0 0 0 7 0 0 0
0 0 1 3 0 6 0 2 0
3 0 6 0 0 1 2 0 0
2 0 0 7 0 0 0 0 0
0 9 0 0 0 0 0 8 7
```

data6

```
0 7 0 8 0 0 0 0 0
2 0 0 3 0 6 0 7 0
0 9 8 0 2 0 1 3 0
0 5 0 0 0 0 9 8 7
8 0 0 0 7 0 0 4 1
9 0 7 0 0 0 0 5 2
0 0 9 0 6 0 5 0 3
7 0 0 1 0 4 0 6 0
0 0 0 0 0 3 7 0 4
```

data7

```
0 2 0 4 0 0 8 0 0
0 0 0 0 1 8 0 7 9
0 8 0 0 7 0 0 0 0
0 0 4 8 6 0 7 0 0
2 0 0 0 4 0 0 9 0
0 7 0 3 0 0 0 0 6
0 0 3 0 8 0 0 2 0
0 0 0 1 0 0 0 8 0
7 1 0 2 0 3 0 0 0
```

data8

```
0 0 0 7 6 8 0 0 0
2 0 0 0 3 0 0 0 0
0 0 6 0 0 9 0 0 4
0 7 0 0 0 0 2 0 0
5 0 0 0 0 0 0 7 8
3 0 1 0 0 0 0 4 0
0 0 9 0 1 0 0 0 2
0 0 0 6 9 3 0 0 0
0 0 0 0 0 5 0 8 0
```

data9

```
0 0 5 3 0 0 0 0 0
8 0 0 0 0 0 0 2 0
0 7 0 0 1 0 5 0 0
4 0 0 0 0 5 3 0 0
0 1 0 0 7 0 0 0 6
0 0 3 2 0 0 0 8 0
0 6 0 5 0 0 0 0 9
0 0 4 0 0 0 0 3 0
0 0 0 0 0 9 7 0 0
```

Source Code

Test.java

```
public class Test {
    static boolean VERBOSE = false;

    public static void main(String[] args) {
        long start=System.currentTimeMillis();

        // use AC3 & backtracking
        //new AC3(Node.getNodes("data9"));

        // use forward checking & backtracking
        new Backtracking(Node.getNodes("data9"));

        long end=System.currentTimeMillis();
        System.out.println("\r\ntotal running time:"+(end-start)+"ms");
    }
}
```

Node.java

```
import java.io.*;
import java.util.HashMap;
import java.util.HashSet;
import java.util.concurrent.ConcurrentSkipListSet;

class Node {
    int row;
    int col;
    int box;
    ConcurrentSkipListSet<Integer> domain;

    Node(int row, int col) {
        this.row = row;
        this.col = col;
        box = calcBox(row, col);
        domain = new ConcurrentSkipListSet<>();
        for (int i = 1; i <= 9; i++) {
            domain.add(i);
        }
    }

    Node(int row, int col, int value) {
        this.row = row;
        this.col = col;
        box = calcBox(row, col);
        domain = new ConcurrentSkipListSet<>();
        domain.add(value);
    }

    static Node[][] getNodes(String filename) {
        Node[][] nodes = new Node[9][9];
        try {
            BufferedReader in = new BufferedReader(new FileReader(filename));
            String line;
            for (int i = 0; i < 9; i++) {
                if ((line = in.readLine()) != null) {
                    String[] numbers = line.trim().split(" ");
                    for (int j = 0; j < 9; j++) {
                        int number = Integer.parseInt(numbers[j]);
                        if (number > 0 && number < 10) {
                            nodes[i][j] = new Node(i, j, number);
                        } else {
                            nodes[i][j] = new Node(i, j);
                        }
                    }
                } else {
                    for (int j = 0; j < 9; j++) {
                        nodes[i][j] = new Node(i, j);
                    }
                }
            }
            printPuzzle(filename, nodes);
            return nodes;
        } catch (FileNotFoundException fnfe) {
            fnfe.printStackTrace();
        } catch (IOException ioe) {
            ioe.printStackTrace();
        }
        return null;
    }

    static void printNodes(Node[][] nodes) {
        System.out.print("\r\n");
        for (int i = 0; i < 9; i++) {
            for (int j = 0; j < 9; j++) {
```

```
            System.out.print(nodes[i][j].printDomain());
        }
        System.out.print("\r\n");
    }
}

private static void printPuzzle(String filename, Node[][] nodes) {
    System.out.println("loading puzzle from " + filename + "...\r\n");
    for (int i = 0; i < 9; i++) {
        for (int j = 0; j < 9; j++) {
            if (nodes[i][j].domain.size() > 1) {
                System.out.print("[0]");
            } else {
                System.out.print(nodes[i][j].printDomain());
            }
        }
        System.out.print("\r\n");
    }
}

// error, return -1
// not end, return 1
// find solution, return 0
static int judgeState(Node[][] nodes) {
    boolean notEnd = false;
    for (int i = 0; i < 9; i++) {
        for (int j = 0; j < 9; j++) {
            if (nodes[i][j].domain.size() == 0) {
                return -1;
            } else if (nodes[i][j].domain.size() > 1) {
                notEnd = true;
            }
        }
    }
    if (notEnd) {
        return 1;
    }

    // check all rows & columns
    HashSet<Integer> rowSet;
    HashSet<Integer> colSet;
    for (int i = 0; i < 9; i++) {
        rowSet = new HashSet<>();
```

```
            colSet = new HashSet<>();
            for (int j = 0; j < 9; j++) {
                rowSet.add(nodes[i][j].domain.first());
                colSet.add(nodes[j][i].domain.first());
            }
            if (rowSet.size() != 9 || colSet.size() != 9) {
                return -1;
            }
        }

        // check all boxes
        HashMap<Integer, Integer> boxesMap = new HashMap<>();
        for (int i = 0; i < 9; i++) {
            for (int j = 0; j < 9; j++) {
                int box = nodes[i][j].box;
                int sum = 0;
                if (boxesMap.get(box) != null) {
                    sum = boxesMap.get(box);
                }
                sum += nodes[i][j].domain.first();
                boxesMap.put(box, sum);
            }
        }
        for (HashMap.Entry<Integer, Integer> entry : boxesMap.entrySet()) {
            if (entry.getValue() != 45) {
                return -1;
            }
        }

        return 0;
    }

    private int calcBox(int row, int col) {
        int box = 0;
        if (row < 3) {
            if (col < 3) {
                box = 0;
            } else if (col < 6) {
                box = 1;
            } else {
                box = 2;
            }
        } else if (row < 6) {
            if (col < 3) {
                box = 3;
            } else if (col < 6) {
                box = 4;
            } else {
                box = 5;
            }
        } else {
            if (col < 3) {
                box = 6;
            } else if (col < 6) {
                box = 7;
            } else {
                box = 8;
            }
        }
        return box;
    }

    @Override
    public boolean equals(Object other) {
        if (this == other) {
            return true;
        }
        if (other == null) {
            return false;
        }
        if (getClass() != other.getClass()) {
            return false;
        }
        Node otherNode = (Node) other;
        if (otherNode.row == this.row && otherNode.col == this.col && otherNode.box == this.box)
{
            return true;
        } else {
            return false;
        }
    }

    @Override
    public String toString() {
        return "(" + this.row + "," + this.col + ")";
    }
    String printDomain() {
        String str = "[";
        for (Integer i : domain) {
            str = str + i + "";
        }
        str += "]";
        return str;
    }
}
```

Backtracking.java (Forward Checking Bolded Below)

```
import java.util.*;

public class Backtracking {
    private int stepCount = 0;
    private Node[][] nodes;
    private Stack<Record> stack = new Stack<>();

    class Record {
        Node node;
        int removedValue;

        Record(Node node, int value) {
            this.node = node;
            removedValue = value;
        }
    }

    Backtracking(Node[][] nodes) {
        System.out.println("\r\nfirst time of forward checking...");

        this.nodes = nodes;
        if (!forwardCheck()) {
            System.out.println("no solution");
            return;
        }
        Node.printNodes(nodes);

        System.out.println("\r\nbacktracking...");
        if (search(getHeuristicNode()) == -1) {
            System.out.println("no solution");
        } else {
```

```
        Node.printNodes(nodes);
    }
}

Backtracking(Node[][] nodes, boolean notUsingForwardCheck) {
    System.out.println("\r\nbacktracking...");

    this.nodes = nodes;
    if (search(getHeuristicNode()) == -1) {
        System.out.println("no solution");
    } else {
        Node.printNodes(nodes);
    }
}

// use recursion to search value for all the nodes
//
// error, return -1
// not end, return 1
// find solution, return 0
int search(Node node) {
    int state = stack.size();

    // 0 or -1
    int result = Node.judgeState(nodes);
    if (result != 1) {
        return result;
    }
    if (node == null) {
        return -1;
    }

    // not end, iterate each value of the node
    int value = getHeuristicValue(node);
    while (value != 0) {
        if (Test.VERBOSE) {
            stepCount++;
            System.out.println("step " + stepCount + "  try " + value + " for " + node.toString());
        }
        setNodeValue(node, value);
        if (!forwardCheck()) {
            // if this value is invalid
            backtrackState(state);
            if (!removeNodeValue(node, value)) {
                return -1;
            }
            state = stack.size();
            value = getHeuristicValue(node);
        } else {
            // if this value is valid
            // search the next node
            result = search(getHeuristicNode());
            if (result == 0) {
                return 0;
            } else if (result == -1) {
                backtrackState(state);
                if (!removeNodeValue(node, value)) {
                    return -1;
                }
                state = stack.size();
            }
            value = getHeuristicValue(node);
        }
    }
    return -1;
}

// establish one node's neighbors arc consistency with it: arc(Y,X)
// when this node has one value in its domain
boolean forwardCheck() {
    boolean isContinue = false;
    for (int i = 0; i < 9; i++) {
        for (int j = 0; j < 9; j++) {
            if (nodes[i][j].domain.size() == 1) {
                ArrayList<Node> neighbors = getNeighbors(nodes[i][j]);
                for (Node n : neighbors) {
                    int value = nodes[i][j].domain.first();
                    if (!removeNodeValue(n, value)) {
                        return false;
                    } else if (n.domain.size() == 1) {
                        isContinue = true;
                    }
                }
            }
        }
    }
    if (isContinue) {
        return forwardCheck();
    }
    return true;
}

// choose one unassigned node with the minimum remaining value
Node getHeuristicNode() {
    ArrayList<Node> nodeList = new ArrayList<>();
    // minimum size start from 2 because all nodes with 1 value in domain are assigned
    for (int size = 2; size <= 9; size++) {
        for (int i = 0; i < 9; i++) {
            for (int j = 0; j < 9; j++) {
                if (nodes[i][j].domain.size() == size) {
                    nodeList.add(nodes[i][j]);
                }
            }
        }
        if (!nodeList.isEmpty()) {
            break;
        }
    }
    if (nodeList.isEmpty()) {
        return null;
    }
    return nodeList.get(0);
}

// assign the selected node with the least constraining value
int getHeuristicValue(Node node) {
    ArrayList<Node> neighbours = getNeighbors(node);
    HashMap<Integer, Integer> countMap = new HashMap();
    for (int value : node.domain) {
        int count = 0;
        for (Node n : neighbours) {
            count += n.domain.size();
            if (n.domain.contains(value)) {
                count--;
            }
        }
        countMap.put(value, count);
    }
    int key = 0;
    int maxCount = -1; // maxCount may be 0
    for (HashMap.Entry<Integer, Integer> entry : countMap.entrySet()) {
        if (entry.getValue() > maxCount) {
            maxCount = entry.getValue();
            key = entry.getKey();
        }
    }
    return key;
}

// remove one value from one node's domain and return true
// return false when the domain is empty
boolean removeNodeValue(Node node, int value) {
    if (node.domain.contains(value)) {
        node.domain.remove(value);
        stack.push(new Record(node, value));
        if (node.domain.isEmpty()) {
            return false;
        }
    }
    return true;
}

void setNodeValue(Node node, int value) {
    for (int i : node.domain) {
        if (i != value) {
            node.domain.remove(i);
            stack.push(new Record(node, i));
        }
    }
}

void backtrackState(int state) {
    while (stack.size() > state) {
        Record record = stack.pop();
        record.node.domain.add(record.removedValue);
    }
}

// get neighbors whose domains have more than 1 node
ArrayList<Node> getNeighbors(Node node) {
    ArrayList<Node> neighbors = new ArrayList<>();
```

```
        // same row
        try {
            for (int i = 0; i < 9; i++) {
                if (i != node.col && nodes[node.row][i].domain.size() > 1) {
                    neighbors.add(nodes[node.row][i]);
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }

        // same col
        for (int i = 0; i < 9; i++) {
            if (i != node.row && nodes[i][node.col].domain.size() > 1) {
                neighbors.add(nodes[i][node.col]);
            }
        }

        // same box
        for (int i = 0; i < 9; i++) {
            for (int j = 0; j < 9; j++) {
                if (i != node.row && j != node.col && nodes[i][j].box == node.box
                        && nodes[i][j].domain.size() > 1) {
                    neighbors.add(nodes[i][j]);
                }
            }
        }
        return neighbors;
    }
}
```

AC3.java

```
import java.util.ArrayList;

public class AC3 {
    private int stepCount = 0;
    private Node[][] nodes;
    private ArrayList<Arc> arcs = new ArrayList<>();

    class Arc {
        Node node1;
        Node node2;
    Arc(Node n1, Node n2) {
        node1 = n1;
        node2 = n2;
    }
}

AC3(Node[][] nodes) {
    System.out.println("\r\nAC3...");

    this.nodes = nodes;

    for (int i = 0; i < 9; i++) {
        addRowArcs(i);
        addColArcs(i);
        addBoxArcs(i);
    }

    while (!arcs.isEmpty()) {
        Arc arch = arcs.remove(0);
        if (!establishAC(arch)) {
            System.out.println("no solution.");
            break;
        }
    }

    int result = Node.judgeState(nodes);
    if (result != -1) {
        Node.printNodes(nodes);
    }
    if (result == 1) {
        new Backtracking(nodes, false);
    }
}

void addRowArcs(int row) {
    for (int i = 0; i < 8; i++) {
        for (int j = i + 1; j < 9; j++) {
            arcs.add(new Arc(nodes[row][i], nodes[row][j]));
            arcs.add(new Arc(nodes[row][j], nodes[row][i]));
        }
    }
}
```

```
void addColArcs(int col) {
    for (int i = 0; i < 8; i++) {
        for (int j = i + 1; j < 9; j++) {
            arcs.add(new Arc(nodes[i][col], nodes[j][col]));
            arcs.add(new Arc(nodes[j][col], nodes[i][col]));
        }
    }
}

void addBoxArcs(int box) {
    // add nodes which are in the same box together
    ArrayList<Node> nodesBox = new ArrayList<>();
    for (int i = 0; i < 9; i++) {
        for (int j = 0; j < 9; j++) {
            if (nodes[i][j].box == box) {
                nodesBox.add(nodes[i][j]);
            }
        }
    }
    for (int i = 0; i < 8; i++) {
        for (int j = i + 1; j < 9; j++) {
            arcs.add(new Arc(nodesBox.get(i), nodesBox.get(j)));
            arcs.add(new Arc(nodesBox.get(j), nodesBox.get(i)));
        }
    }
}

boolean establishAC(Arc arc) {
    Node node1 = arc.node1;
    Node node2 = arc.node2;
    for (Integer i : node1.domain) {
        boolean isDiff = false;
        for (Integer j : node2.domain) {
            if (i != j) {
                isDiff = true;
                break;
            }
        }

        if (!isDiff) {
            node1.domain.remove(i);
            if (Test.VERBOSE) {
                stepCount++;
                System.out.print("step " + stepCount +
                        "  set.size():" + arcs.size() +
                        "  AC:" + node1.toString() + node2.toString() +
                        "  " + node1.toString() + ".remove(" + i + ")" +
                        "  domain:" + node1.printDomain() + "\r\n");
            }
            if (node1.domain.isEmpty()) {
                return false;
            }
            addNeighborArcsExceptB(node1, node2);
        }
    }
    return true;
}

void addNeighborArcsExceptB(Node A, Node B) {
    // add arcs with nodes in the same row
    for (int i = 0; i < 9; i++) {
        if (i != A.col && nodes[A.row][i] != B) {
            arcs.add(new Arc(nodes[A.row][i], A));
        }
    }

    // add arcs with nodes in the same col
    for (int i = 0; i < 9; i++) {
        if (i != A.row && nodes[i][A.col] != B) {
            arcs.add(new Arc(nodes[i][A.col], A));
        }
    }

    // add arcs with nodes in the same box
    for (int i = 0; i < 9; i++) {
        for (int j = 0; j < 9; j++) {
            if (nodes[i][j].box == A.box && i != A.row && j != A.col && nodes[i][j] != B) {
                arcs.add(new Arc(nodes[i][j], A));
            }
        }
    }
}
}
```